

Running Gunicorn and MOP locally

Executive Summary

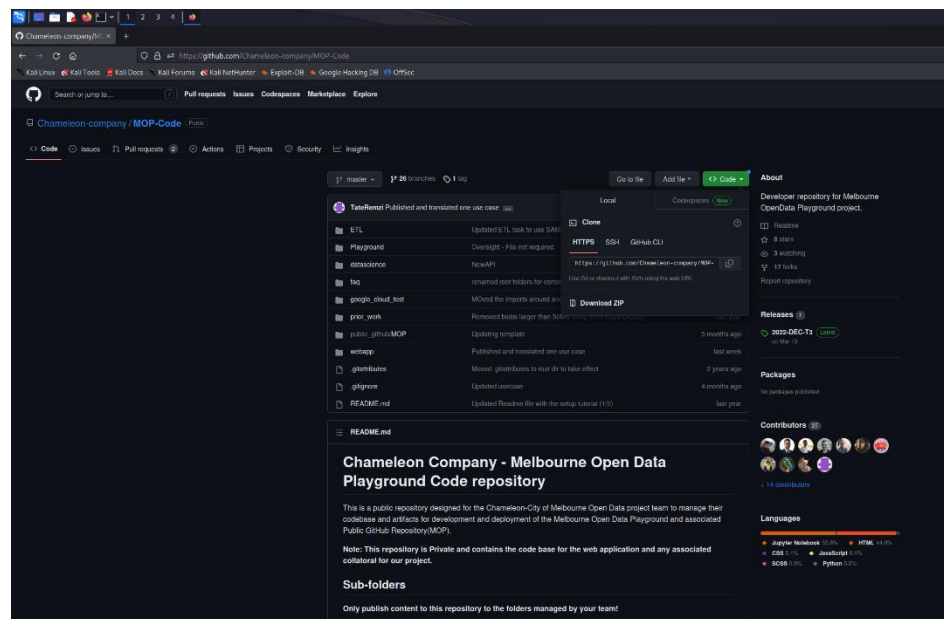
Following from my previous task where I reported security vulnerabilities, I thought I would try to implement these fixes myself, given I have read over the documentation in learning how it works and wanted to mend these errors myself.

Unfortunately, I have not succeeded in this sprint's task in mending these fundamental vulnerabilities, primarily because of the effort to emulate the server locally to match the environment of the currently public website as a verification step to my fixes.

So as part of this sprint's deliverables I would like to share what I have created for my given task: a local version of the project site running on accurate systems (Gunicorn and certificates) on Debian operating systems (in my case Kali Linux, but extends to ParrotOS).

Step 1: Download MOP source files

Importantly we need the source code locally on our machine. To do so, we will log into our GitHub account within the web browser and download the zip file there, and unpack them.



Lucas Kocon

218510242

01/05/2023

Step 2: Installing Python

Surprisingly Python does not release Linux or Debian installers, which is annoying given the MOP project works entirely on it. I found this website (linked below) and ran through those steps to get Python installed on Kali Linux. This will apply to any other Debian systems that future security members might use (such as ParrotOS which other members have used during this trimester of work).

<https://cloudinfrastructureservices.co.uk/how-to-install-python-3-in-debian-11-10/>

While you can follow the steps on the website to installing Python, I will relay those steps here. All the steps are done within the command line:

1. **sudo apt update** – this updates the Debian local package index so it can find an up-to-date Python installer and its dependencies.
2. **sudo apt install build-essential zlib1g-dev libncurses5-dev libgdbm-dev libnss3-dev libssl-dev libreadline-dev libffi-dev libsqlite3-dev wget libbz2-dev** – these will install all of Debian's dependencies needed to install Python.
3. **wget <https://www.python.org/ftp/python/3.11.1/Python-3.11.1.tgz>** - this is the install location of Python's Ubuntu installer and will download a compressed installer file. For some reason this link or the associated file are not available publicly, like they are for the Windows installers (even when viewing the website on a Debian machine).
4. **tar -xvf Python-3.11.1.tgz** – this uses Debian's in-built archive system to unpack the files downloaded.
5. **cd Python-3.11.1** – enters the directory we just unpacked.
6. **sudo ./configure --enable-optimizations** – checks if the dependencies are in place and tries to optimise the files for installation.
7. **sudo make -j 2** – doing this command will compile the Python code onto the system. This will take a while.
8. **sudo make altinstall** – this is the final critical step that installs all the compiled code. Afterwards Python will be installed and available to run on the system.
9. **python3.11 -V** or **python -V** – to make sure Python is working, run these prompts to return the Python version.

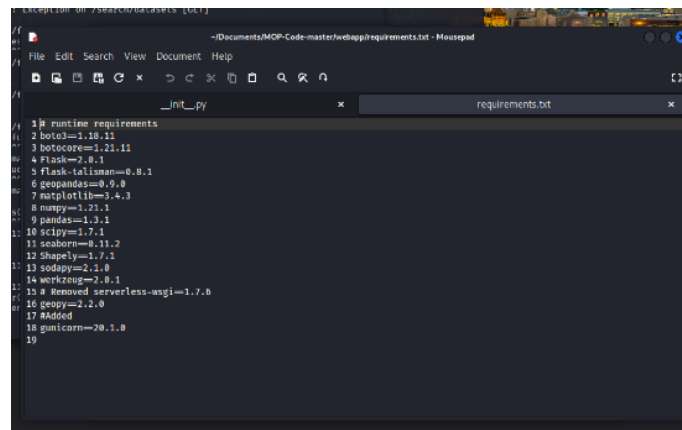
Lucas Kocon

218510242

01/05/2023

Step 3: Install all Python Libraries

The MOP project uses many Python libraries to function, which handily provides a list to check through before running the project. Run through all these pip installations (**pip install**) beforehand so running the project will not be halted. You can either run through every list item provided in the requirements document in the project.



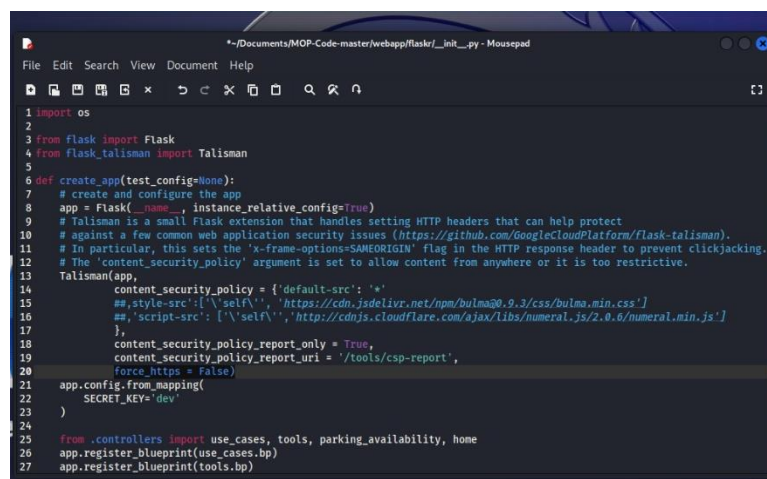
```

1 # runtime requirements
2 boto3==1.18.11
3 botocore==1.22.11
4 flask==2.0.1
5 flask-talisman==0.8.1
6 geopandas==0.9.0
7 matplotlib==3.4.3
8 numpy==1.21.1
9 pandas==1.3.1
10 scipy==1.7.1
11 seaborn==0.11.2
12 Shapely==1.7.1
13 sodapy==2.1.0
14 werkzeug==2.0.1
15 # Removed serverless-ugui==1.7.0
16 geopy==2.2.0
17 #Added
18 gunicorn==20.1.0
19

```

Another method is to just install Flask (**pip install Flask**), run the project (**python -m flask run** while in the project directory) and install whatever library halts the program from running each time. This can still run the program, but there are a few issues where we cannot see the website.

That will be because the project is configured for a forced HTTPS connection but there are no certificates being used, so the HTTP requests aren't recognised and the site won't load in a browser. We can mediate this by going to the "MOP/webapp/flaskr" directory, opening the "__init__.py" file in finding the Talisman object being created, we can create a field within the Talisman object and configure it so it doesn't force a HTTPS connection, allowing a HTTP connection to be made instead and the site can be seen in the browser.



```

1 import os
2
3 from flask import Flask
4 from flask_talisman import Talisman
5
6 def create_app(test_config=None):
7     # create and configure the app
8     app = Flask(__name__, instance_relative_config=True)
9     # Talisman is a small Flask extension that handles setting HTTP headers that can help protect
10    # against a few common web application security issues (https://github.com/GoogleCloudPlatform/flask-talisman).
11    # In particular, this sets the 'x-frame-options=SAMEORIGIN' flag in the HTTP response header to prevent clickjacking.
12    # The 'content_security_policy' argument is set to allow content from anywhere or it is too restrictive.
13    Talisman(app,
14             content_security_policy = { 'default-src': '*'
15                                     #, 'style-src': ['\self', 'https://cdn.jsdelivr.net/npm/bulma@0.9.3/css/bulma.min.css']
16                                     #, 'script-src': ['\self', 'https://cdn.jsdelivr.net/npm/bulma@0.9.3/css/bulma.min.css']
17             },
18             content_security_policy_report_only = True,
19             content_security_policy_report_uri = '/tools/csp-report',
20             force_https = False)
21    app.config.from_mapping(
22        SECRET_KEY='dev'
23    )
24
25    from .controllers import use_cases, tools, parking_availability, home
26    app.register_blueprint(use_cases.bp)
27    app.register_blueprint(tools.bp)

```

At this point, it is already better to run the project because a different gateway brought into the project files will run and all local JavaScript will be used thus allowing a lot more of the project's functions like data analysis and presentations to be used and tested; when hosting the site on Windows the gateway service will not run, the JavaScript would not load and none of these services can be used. But, I do not know how this gateway service works and Gunicorn is listed as part of the project's tech stack (and thus how the project publicly available has been deployed), so we will try to run it there.

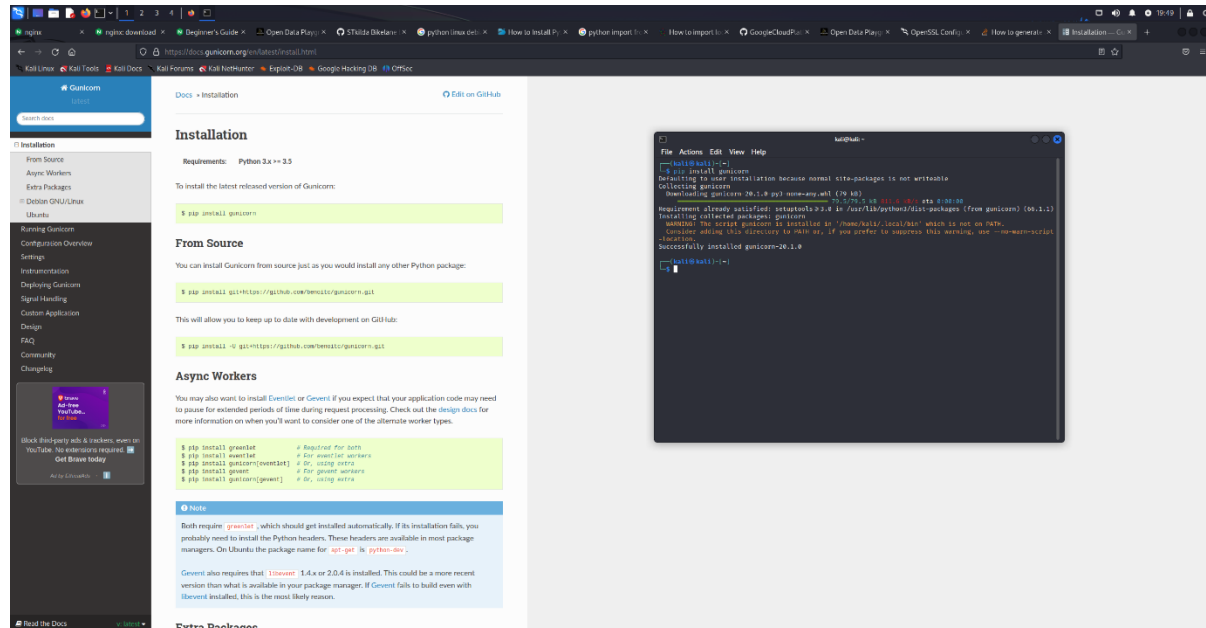
Lucas Kocon

218510242

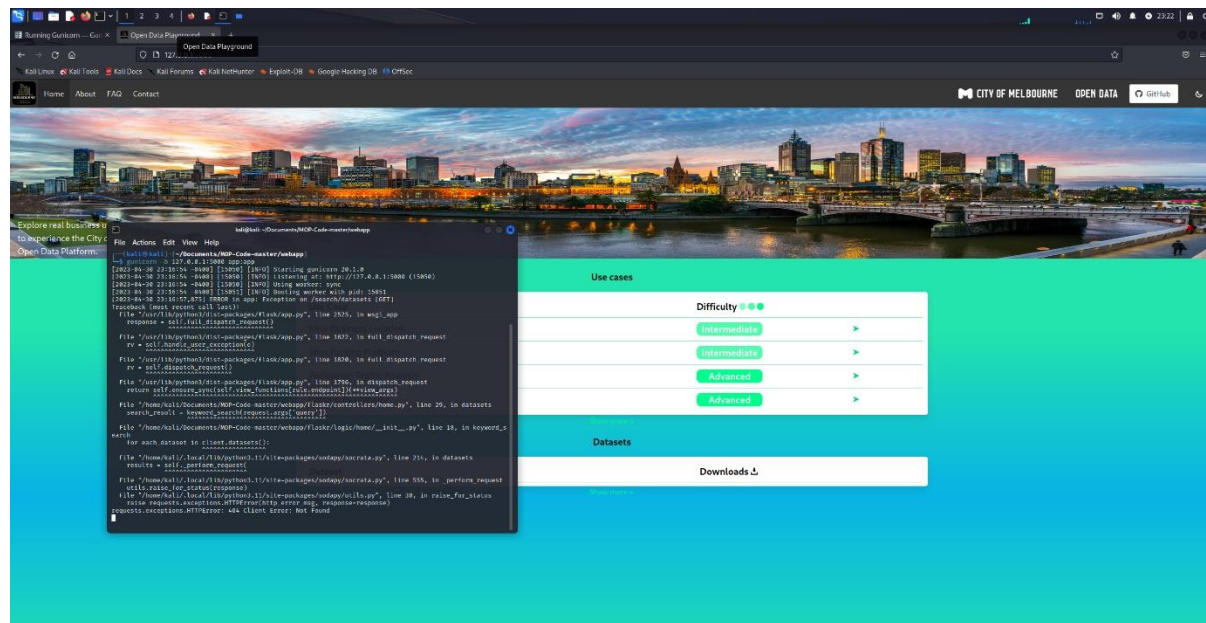
01/05/2023

Step 4: Install and Run Gunicorn

To use Gunicorn, we shall install the Python library as described on the website.



With Gunicorn installed, we can enter the command “**gunicorn -b 127.0.0.1:5000 app:app**” and host the server. We can change the IP address portion for not only a different address but also a different port. Also we can similarly only run if the connection isn’t forced onto HTTPS. The “**app:app**” portion is critical to running from the correct file, and the file we need is aptly titled “**app.py**” sitting within the “**webapp**” directory; were it to be called something else like “**main.py**” later during development, this part of the command is to be adjusted and the project should work as normal.



From here we have matched the product’s deployment environment for testing during development and security. This can be a safe spot to use the environment, but for completeness we shall use certificates for any security analysis that we will employ.

Lucas Kocon

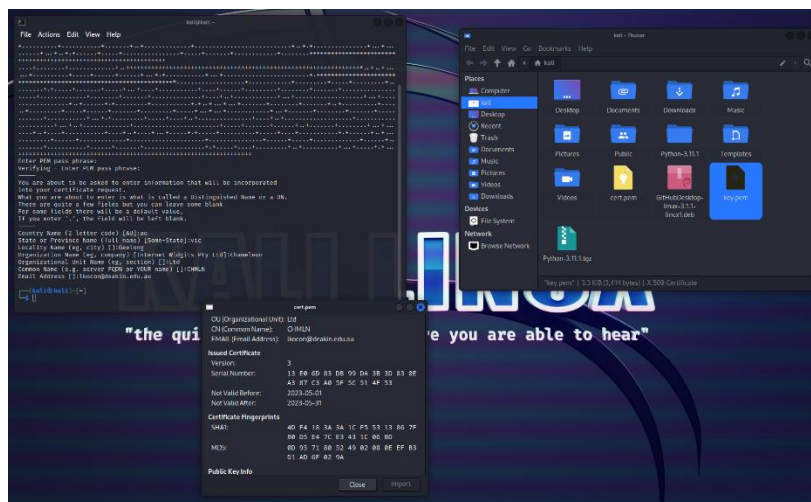
218510242

01/05/2023

Extra Step 5: Install OpenSSL and Using Certificates

Security systems like Kali Linux already have Kali Linux installed so we can use it right away. While OpenSSL lists all its options when running the `--help` command, it can be overwhelming given it does not say what each option does and how we can create a certificate for HTTPS communication to our project. Note: if you see a public site that uses a self-signed certificate (which you will see how your browser will warn you during this step), you can expect a supremely dodgy website that normally should be avoided

So I read this Stack Overflow post (<https://stackoverflow.com/questions/10175812/how-to-generate-a-self-signed-ssl-certificate-using-openssl>) and sourced its top answer in creating a self-signed certificate. To make these certificates, we enter into the command line **`openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -sha256 -days 365`**, which will create a certificate and a 4096-bit RSA key (at the location of our current directory). OpenSSL will require a password for the **`key.pem`** file; keep that password in mind because we will need it for something else. What we have created is a certificate named **`cert.pem`** and an encrypted RSA key file **`key.pem`**.



To make the Gunicorn environment work with these files, we will update the command to **`gunicorn -b 127.0.0.1:5000 --keyfile key.pem --certfile cert.pem app:app`**. This will launch the app with these certificates when we want to view the site through HTTPS in unenforced (and if we revert the change in Python to enforce this), but for the first time the browser will warn of you viewing a site with a self-signed certificate.

However there is a noticeable issue: the browser loads but doesn't display anything, and if you look at the terminal it will ask for the PEM password – the bigger issue is it will keep asking for the PEM password. This is very annoying because it will keep asking for multiple transmissions and there is no good visual feedback in the browser. What has happened is that Gunicorn has kept asking for this password on the **`key.pem`** file, because the content is the encrypted private key for the RSA scheme. I do not know what is the proper solution in automatically entering this phrase for an autonomous server, but given we are only hosting this site locally for development and security checks we can use a cheap and nasty workaround.

Lucas Kocon

218510242

01/05/2023

I also found this article that involves our OpenSSL-created certificate

(<https://help.cloud66.com/docs/security/remove-passphrase>) and the solution involves this command: **"openssl rsa -in key.pem -out keynop.pem"**. We will be prompted with a PEM password, but the output in **"keynop.pem"** is that not only will we not be prompted when we read the file for ourselves, running the project with Gunicorn with this key file will stop prompting us about the PEM password too; this is because the new file is unencrypted and contains the private key needed for our certificate, so no PEM password is necessary.

And after all this, we have successfully created a simulated Gunicorn environment to test if development and security changes will work on the site, and hopefully it can be a handy simulation for pen-testing without risk to the deployed site.

